After deciding on a textual based approach as explained in chapter 1. We went through a short evaluation process to decide the best solution for creating the opus domain specific language (DSL).
The first choice was between internal DSLs vs. external ones. Internal DSL means a DSL that is developed inside of a language. The advantage of an internal DSL is that since it uses a language the language compiler/interpreter already knows all the constructs and it is easy to develop code generations etc. on top of it.
The downsides of internal DSLs is that the syntax is bound by the language capabilities for instance Java is very verbose and limited in its extension capabilities so a DSL in Java usually boils down to creating a "fluent interface" (http://en.wikipedia.org/wiki/Fluent_interface) which is based on method chaining e.g.

```
Vacation vacation = vacation().starting("10/09/2007")
                              .ending("10/17/2007")
                              .city("Paris")
                              .hotel("Hilton")
                              .airline("United")
                              .flight("UA-6886");
```

To make the DSL more rich we can use languages that allow more advanced semantics like Scala or Python and express more about our objects for example the following is a snippet for an internal DSL in Scala for defining Events:

```
case class SubscriberEvent(a:Key[Long],
                 b:FK[Int],
                 c: String,
                 msisdn : Key[Telephone]    ,
                 anotherField : Enum[String]) extends Event[SubscriberEvent] {
  msisdn partitionBy PartitionStrategy.Prefix
  a partitionBy PartitionStrategy.Random
  b relatesTo classOf[Customer]
  msisdn format "XXX-XXX-XXXX"
}
```

The second, more significant, problem of internal DSLs is that it is part of the language so there is no way to limit the use of the language in between the DSL constructs – this can be very confusing for people getting started using the language and in any event makes it hard to validate and provide tooling around the DSL

The alternative to internal DSLs is an external DSL. In external DSLs the syntax includes just the constructs and concepts you define which makes it very focus. The downside of external DSLs is that you have to develop all the parsers, editor (to provide validation, syntax highlighting) and generators to make it usable. Fortunately, there are several products in the market the make the development of external DSLs easier and provide the basic capabilities out of the box. These set of tools are known as Language Workbenches. During the evaluation that resulted in choosing XText we looked at the following options:

- Python's pyparsing http://pyparsing.wikispaces.com/
- XText http://www.eclipse.org/Xtext/index.html
- Jetbrain's MPS 3.0 http://www.jetbrains.com/mps/
- Rascal http://www.rascal-mpl.org/
- Spoofax - http://strategoxt.org/Spoofax
- Whole http://whole.sourceforge.net/

Python's pyparsing is just a parser and does not provide editor capabilities so it was a low priority option. Spoofax & Whole are both LGPL so were disqualified as using problematic license. Which left us with MPS, Rascal and XText.

MPS looks very impressing at first sight. When creating a new project it creates a tree structure where you can specify all the needed parts of a language (see Figure 1 below).
However MPS suffers from three main problems:

- It is very cumbersome to work with – creating concepts takes a lot of time. The editor causes a lot of friction (for comparison 2 days of work in MPS were recreated in 30 min using XText)
- It uses projectional editing – i.e. it isn't really an editor it is a structural editor where you basically "fill in the blanks" in a template. It is very weird working with
- We cannot extend it with graphical editors – it is purely textual and we like to add graphical editor for at least some of the concepts

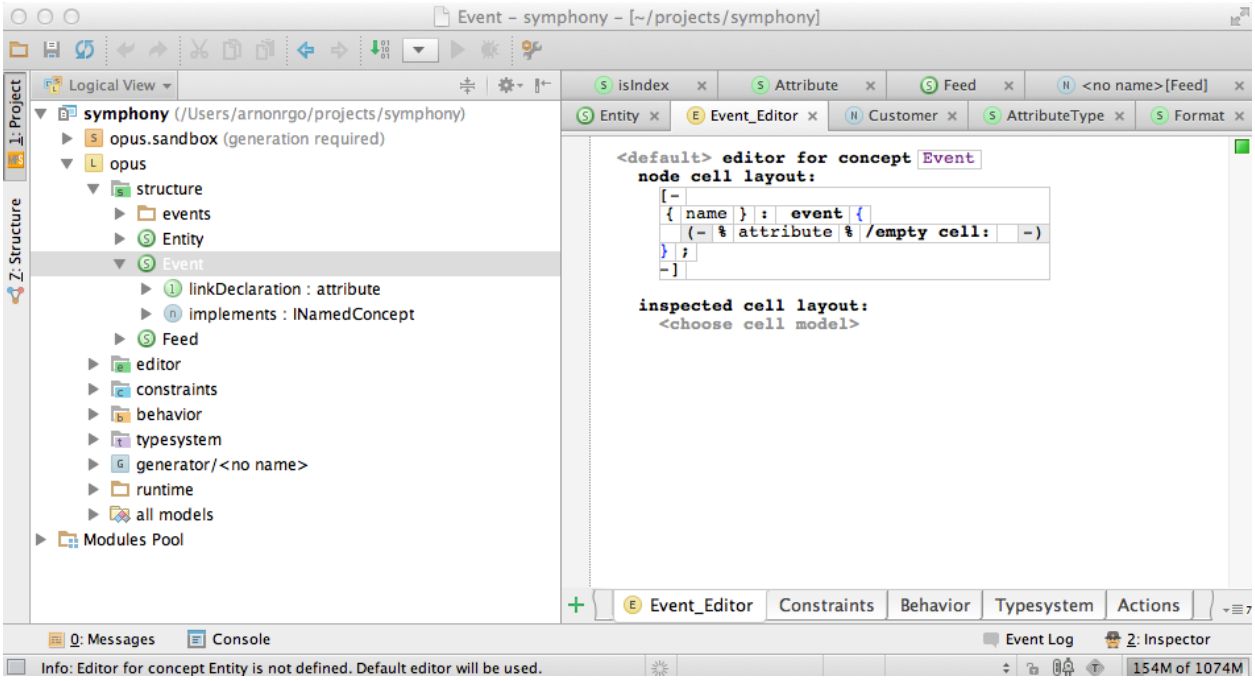

Figure 1 - MPS editor

Disqualifying MPS left us with Rascal and Xtext – both of them have eclipse plug-in and are relatively easy to work with. XText is the preferred solution however since it

- Built around Eclips Modeling Framework (EMF) so it integrated well with other eclipse tooling for code generation – e.g. we can easily extend it with graphical editors like Serius (http://eclipse.org/sirius/getstarted.html) or Spray (https://code.google.com/a/eclipselabs.org/p/spray/)
- It has very comprehensive integration with eclipse and so it allows creating very sophisticated editors (with quickfixes, rules, templates, wizards etc.)
- Overall it was easier to work with compared with Rascal (and any other option for that matter)